POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

TinyML on-device neural network training.

Tesi di Laurea Magistrale in
Computer Science Engineering - Ingegneria Informatica

Eugeniu Ostrovan, 10527025

Advisor:
Professor Manuel Roveri Ph.D.

Co-advisors:
Massimo Pavan

Academic year:
2021-2022

Abstract: Tiny Machine Learning (TinyML) is the research field that aims to join the high representational power of machine learning solutions with the tight hardware constraints imposed by embedded/IoT devices.
A few hundred kilobytes of RAM are available on tiny hardware, and the microcontroller unit's (MCU) clock rates are in the order of the KHz. Such scarce resources make deploying deep learning solutions such as feed-forward neural networks to the device challenging. Nevertheless, remarkable results have been achieved, for example, in keyword spotting.
The usual pipeline for these solutions is to first train the algorithms on highly performant hardware. Then, convert the solution for on-device inference in a second step.
On-device learning, instead, refers to the ability to train and adapt models directly on embedded and edge hardware. Currently, on-device learning is not supported by any commercial tinyML frameworks for non-specialized hardware. The academic research works in this field are still few and mainly focused on specific problems.
This thesis develops a toolbox for on-device neural network training that can convert input models into an embedded format with training capabilities. The model architectures that the toolbox is targeting are dense feed-forward neural networks (FFNN) and convolutional neural networks (CNN).
The solution has been tested on standard datasets for tasks of on-device training, transfer learning, and incremental learning after a concept drift.

Key-words: tinyml, machine learning, embedded devices

1. Introduction

Artificial intelligence is a multidisciplinary study to describe and simulate learning and other features of intelligence.[7] A significant area of artificial intelligence is machine learning, the paradigm of programming information processing systems based on learning programs from data and respective targets. Machine learning is opposed to the classical machine programming setup in which handcrafted programs are applied to data to produce results.[12]

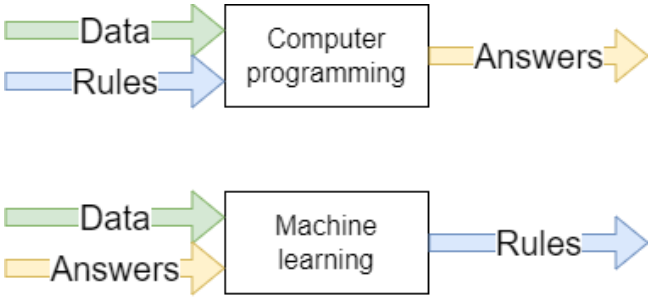


Figure 1: Comparison between computer programming and machine learning paradigms

Machine learning has been successful in the last decade in solving numerous modern problems thanks to the availability of large and high-quality datasets, improvements in hardware technology and algorithms. Neural networks are one of the areas of machine learning. These, in particular, have been widely adopted since they can be used to generate useful data representations for solving the problem at hand without the need for manual feature engineering. It is possible to extract useful features from data even when it is not labeled.[23] Some problems with which machine learning has had great success include: image recognition, classification, text translation, speech recognition as well as playing complex games like chess, go, or starcraft. Today artificial intelligence methods are not limited only to academic research. They have also found commercial use: recommender systems, voice interfaces, predictions, and insight extraction from data to drive business decision-making. The main focus in pushing AI state-of-the-art has been increasing model accuracy and learning efficiency. Today more subtle aspects of solution design are starting to become very relevant: explainability, scalability, and mainly for our purposes, efficiency.

1.1. TinyML

Tiny machine learning is a recent research field focusing on reducing the computational resources required for machine learning solutions and making it possible to deploy them to resource-constrained embedded devices.
In order to create machine learning solutions, data is transferred to higher power computing infrastructures such as workstations or the cloud to build models and solutions. However, it is known that data transmission requires an order of magnitude more energy than information processing or memory access.[19] Data transfer also has the limitations of bandwidth and reliability besides energy efficiency. It can be impossible to transmit all the data collected by the device or do so promptly to guarantee acceptable latency.
High power computing is essential for modern machine learning success. However, a different class of computing infrastructures that has not received much attention is that of edge and embedded devices. Resource-constrained and specialized devices are energy-efficient, massively distributed (23.5 billion microcontrollers shipped in 2020), and cheap. These are ideal for a data-centric approach to AI solutions.

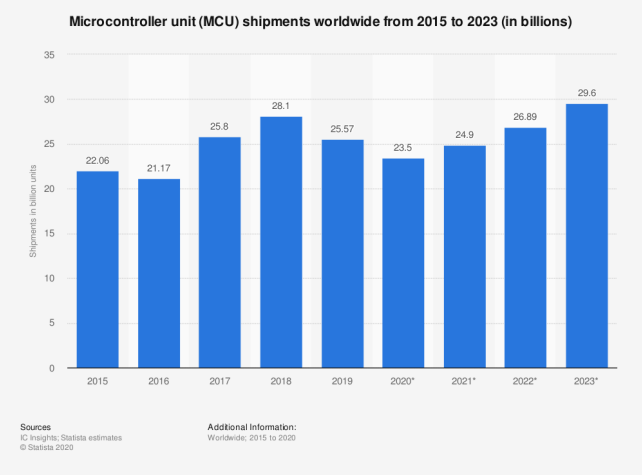


Figure 2: Number of microcontroller units shipped over the years

One application that has driven progress in this field is monitoring audio input for speech recognition[37]. The problem of recognizing speech can be solved with modern machine learning. However, a voice user interface requires that we process input constantly. That implies a considerable energy expenditure. In the case of deployment to smartphones, this is a problem because battery life is limited. Furthermore, most of the input to be processed does not contain meaningful information. Therefore, we only need to apply the high power processing to the input parts where the user is speaking.
Researchers at Google have solved this problem by introducing a cascade architecture composed of one module designed to detect the input segments to which the system needs to respond[37]. This first module is characterised by low power consumption, very high energy efficiency, and it is always on. The second module is the high power system which can respond to the input, and it is woken up by the first module only when needed.
Microcontrollers are the perfect computing infrastructure for energy efficient, always on and distributed solutions. Embedded devices have been considered incompatible with machine learning due to their significant limitations of computational resources. The arduino nano 33 BLE sense has an ARM Cortex M4 MCU running at 64MHz and only 256kB SRAM[1]. It has been shown, however, that the excellent performance of deep learning solutions can be reproduced with a significant reduction of the compute power. This means that given some solution S with performance P on some task, it is very often possible to find a solution S' with comparable performance, but significantly reduced resource requirements[26].
Network compression is one of the methods with which this can be achieved. Techniques for network compression include SVD decomposition, network pruning, and quantization[26]. All these techniques apply principally to model evaluation, that is calculating the response of a model, and they offer a compression significant enough to enable the deployment of deep learning solutions to smartphones and even to microcontroller powered devices.
Another front driving progress in TinyML is the design of efficient and specialized hardware: TPU[21], analog processors[38], but also hardware accelerators[15]. These allow to push the limits of what tiny devices can accomplish.
Significant progress is also being made with the development of efficient code for making better use of the available hardware. CMSIS-NN is a library providing very efficient implementations of neural network kernels for ARM Cortex CPUs.
Training of tiny models is a more difficult problem on the other hand. This is due to the high memory requirement for storing intermediate activations[11, 18]. Pruning and precision reduction are not as effective in the context of training. Pruning reduces the degrees of freedom and flexibility of the model, making it much harder to learn. Precision reduction can have a very bad effect on training, because weight updates rely on precise derivative calculations. Nevertheless progress is being made in this area as well[20].
A different application of TinyML is edge processing. When data is gathered on edge nodes it has to be transmitted to a central node or the cloud in order to be processed. Data transmission is very energy intensive, an order of magnitude more than data processing[]. Local elaboration therefore would enable significant energy savings and latency reduction between input collection and response generation. Furthermore data transmission is also limited in terms of bandwidth, network availability and transmission reliability. TinyML opens the path to improvements in reliability, security and data efficiency and it is overall a data centric approach.

1.2. Problem statement

The main goal of this thesis is to develop on-device learning. A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.[25]
We start from the well established TensorFlow lite micro framework for TinyML which currently does not support training. We reason how such functionality could be added by considering an example of a practical application and the constraints that make implementing such functionality difficult.
The practical application is a CNN applied on UWB doppler radar data for presence detection and classification in a car[9]. Accuracy results in the range of 0.9 are achieved. In analysing how this result can be further improved we have noticed that there are noticeable differences in the data gathered in different vehicles, because they have different shapes and sizes. Data is also sensitive to the radar placement: small changes in the radar position or direction can have noticeable impact on model's performance. Furthermore data collection in this particular environment is slow and expensive, making it prohibitive to gather a big enough dataset to generalise over all these differences. Factors that have made data collection difficult are the need to collect samples in many different makes of vehicles and the fact that we are working with people, children and pets, therefore the data collection process cannot be easily automatised. These problems are reminiscent of the situations in which transfer learning has found fruitful application and we will come back to it.
The constraints which make training ML models on tiny devices are the little processing power available (soft constraint) and, most importantly, the limited memory (hard constraint). An analysis of how the memory is used in the process of using and training neural networks shows that most of it serves for storing intermediate activations which are needed for calculating the error derivatives during backpropagation[30]. Starting from literature insights and theoretical knowledge we have developed a tool to measure the memory required for evaluating and training a network as well as the amount of processing (measured in number of operations). Values are calculated at the granularity of individual layers.
Transfer learning works by taking a model from a source domain, freezing some of it's weights and adapting with data from the target domain. This is the inspiration for our solution: given a model, it's feature extraction block is deployed using TFLM, it is therefore frozen. For the head of the model, on the other hand, we generate the code that implements it and permits training.
Once we develop the on-device learning functionality we move on to its applications and test it for more complex problems: incremental learning, transfer learning, and concept drift.
In this thesis we develop on-device learning for TinyML, furthermore, we show it would improve the quality of the solution in terms of accuracy by allowing a greater degree of customisation to the specific environment of deployment.

1.3. Datasets and evaluation

The solution is evaluated by comparing performance to baseline TFLM deployment. The learning capability is also evaluated by comparing it to a different solution from literature, Train++ developed at the School of Electronic Engineering at the Dublin City University. A second hypothesis of our work is that great value can be obtained by the application of machine learning solutions to radar data, so we evaluate our solutions on UWB doppler radar data. Furthermore we evaluate solutions using mul tiple datasets: banknote authentication[2], IR-WBR[5], MNIST[6], fashion MNIST[3]. Tests include the estimation of memory occupation, measurement of computation time, and accuracy.

2. Background

2.1. Incremental learning

Incremental learning is a learning paradigm opposed to batch learning. It refers to the practice of gradually training a model as soon as new labelled data become available. This can be interpreted in two ways: in the first, the task is well defined and static while data becomes increasingly available during training[34]. This is also known as continual learning and the main problem it presents is interference since the continually fed data cannot be assumed independent and identically distributed. Under the umbrella of incremental learning problems a second class is task incremental learning, the scenario in which the definition of the task to be solved by the model evolves over time, for example by the addition of new classes[24]. Some of the challenges in this setup is the requirement to have a dynamic model structure as well as the fact that as the number of classes increases they are more likely to be similar and therefore harder to distinguish accurately.
The established methods for incremental learning can be grouped into two categories. The first is rehearsal methods which involve recalling past instances in order to recover the loss of performance on old tasks. The second category involves the use of regularisation methods in order to limit the changes to the weights that would produce bad interference. Knowledge distillation is traditionally used to compress a large model or an ensemble into a single smaller model. In the context of incremental learning this method can be applied between successive iterations of a model in order to maintain comparable activations on past tasks when learning the new one.

2.2. Catastrophic forgetting

One fundamental problem for incremental learning is catastrophic forgetting, which has been widely discussed in the literature.[17] Catastrophic forgetting, also known as interference, describes the behaviour observed in trained neural networks when these are adapted to a new task, a different environment or variations in the data distribution. When adapting to such a new context the network's performance on the previous task can degrade significantly. As the network learns to solve the new task, it forgets the old one. This issue can also be viewed as difficulty to generalise, to create a solution that provides satisfactory performance on both tasks. This is a problem, because it limits the complexity of the tasks that can be solved by the network. If a solution is excessively specialised, it loses much of its utility in real world application.
As an example consider a convolutional neural network trained to recognize human faces and whether they are wearing a mask or not. If trained on a limited dataset there might be a dependence on lighting conditions so when the solution is deployed we might observe good performance in the morning and significantly reduced performance in the evening. More likely we might observe a difference in performance over the scale of the subjects in the image: faces too far away or too close might be harder to recognize and classify.
Having observed this behaviour one intuitive solution is to train the network using data representative of the instances on which it performs poorly, this however can result in a loss of performance on the instances on which the network initially performed well. This is catastrophic forgetting.
To impress a more intuitive grasp, consider the following experiment on the trivial dataset of addition facts. A network learns one's addition facts. After that it learns two's addition facts. This is the curve of the network's performance on one's addition facts and two's addition facts which shows catastrophic forgetting
The network loses 70% of its performance on the initial task as soon as the performance on the new task becomes marginally better than random and forgets the initial task completely when the second one is fully learned.
Catastrophic forgetting can also be framed as the plasticity-stability dilemma. Training a neural network involves modifying its parameters and requires the flexibility to mould the network to the patterns relevant to the task at hand. When training on new data or for a new task, the process can overwrite parameter values which are essential to correctly solving previously seen tasks and samples and consequently forgetting knowledge learned in the past. An ideal solution would be plastic enough to be always able to integrate new information while remaining stable at the same time in terms of performance on all previously encountered data and tasks. On the other hand it is not always desirable to maintain knowledge of past experience, since it might not remain perpetually relevant.
It has been shown that the forgetting phenomenon applies not only in the case of big changes such as domain transfer and task expansion, but also to a lesser degree between individual batches of samples. In fact, widespread learning algorithms exhibit a notable bias towards the most recent data, particularly in the layers closest to the output. Error backpropagation achieves the best results when training with multiple passes over batches of independent and identically distributed samples. This allows the network to see a wider range of the task input domain and prevents it from fitting too close to any feature that is too specific and would hinder generalisation.

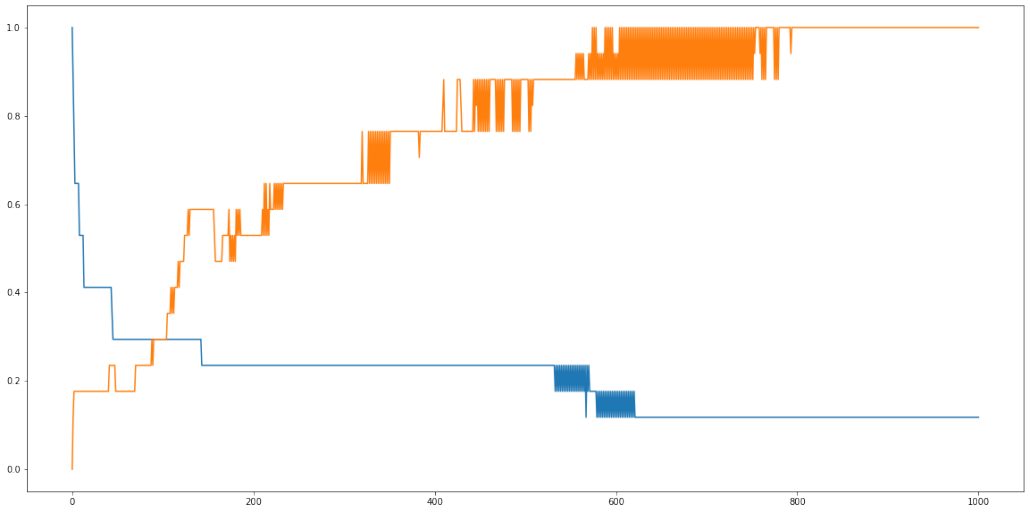


Figure 3: Model accuracy as a function of training epochs on the new task. Blue: first task. Orange: second task

2.3. Data augmentation

In the particular case when the network uses hierarchical convolutions to extract features there is invariance to spatial translations, but not necessarily to scale or rotation. If the dataset presents the same feature at different scales or rotations, the convolutional neural network would learn the features as separate, instead of the same one under a simple transformation. A universally used technique to improve the generalization of features under simple transformations is data augmentation. Data augmentation consists in feeding the network the same samples repeatedly, but after some transformations such as rotation, translation, scale, shear. The effect of this technique is to force the network to learn features that are independent of transformations that we deem not to be informative to the task at hand. The application of data augmentation results in better generalization performance as well as increases the number of samples available for training which allows to create more complex networks. A more relevant example that results in more acute forgetting and which cannot be easily handled using more complex networks or data augmentation are the cases of transfer learning and incremental learning.

2.4. Concept drift

In a real-world setting problems that we solve with our models might depend on some hidden concept or features that cannot be measured. Changes in this hidden concept can induce a change in the problem targets, this is what we refer to as concept drift[32]. A different class of drift is the phenomenon in which the problem definition remains static, but the data distributions or the relevant features change. Both of these happen mainly because of the evolution of the environment.
In this scenario is it desirable to forget past data if it allows a higher degree of flexibility for adapting the features and the rules learned to solve the task at hand to the new context. Forgetting is not a generally negative phenomenon and it is not always trivial to detect concept drift. The difficulty lies in distinguishing between noise and true concept drift. The combination of these factors makes the plasticity-stability trade-off and catastrophic interference challenging problems.

2.5. Transfer learning

Transfer learning is a technique used to apply a network trained on some specific source domain to a target domain. The main reasons for applying transfer learning is the lack of specific data in the target domain. The technique consists in adapting the network from the source domain by fine tuning to the target domain. This works under the assumption that the source and target domains share common features. One example of transfer learning is using the VGG feature extraction block trained on the imagenet dataset to create a Unet model for the segmentation of images of plants. Even though imagenet does not have specific knowledge of our target domain, from its images it is possible to train the network to extract universally useful features such as corners, edges, shapes and various combinations thereof.

2.6. Tensorflow lite micro

Tensorflow Lite Micro(TFLM)[14] is a framework designed to run neural network models on small, low powered computing devices. It offers support for a subset of the TensorFlow operators and allows to convert TensorFlow models into a format suitable for microcontrollers. The framework does not require operating system support, the standard libraries or dynamic memory allocation, making it optimal for deployment on embedded devices. TFLM is composed of two modules: the converter and the interpreter. Given a TensorFlow implementation of an input model, the converter produces the respective model in tflite format which can in turn be converted into a hex string that can be uploaded on the tiny device. The interpreter, on the other hand, runs on the microcontroller itself and uses the model definition produced by the converter to execute inference. Training is not supported by TFLM. The lite model format is a flatbuffer[4] data structure, a serialised object which can be read without unpacking, making it fast and efficient with the downside that it cannot be modified. Modifying a flatbuffer requires deserializing the object, updating the values and then creating a new flatbuffer.
Memory is the most stringent constraint in TinyML, for the reduction of memory required by the model TFLM supports prunning and quantization. Weight quantization can be performed without any further inputs, but activation quantization requires examples of input values. These examples have to be representative of the whole range of inputs that the network will work on. They are required in order to estimate the range of values of the activations. The example data used for quantization is provided to the TFLM converter using a generator function.

```
def yield_representative_dataset():
  for data in tf.data.Dataset.from_tensor_slices
  ((train_generator[0][0].reshape(32,64,64,3))).batch(1).take(10):
    yield [tf.dtypes.cast(data, tf.float32)]
```

2.7. Pruning

Pruning is a technique used to reduce the size of networks. It consists in cutting weights and neurons from the network with the effect of reducing the memory required for storing the model as well as reducing the amount of computations needed to evaluate it. This also has an effect on performance since by cutting away some weight, neuron or filter some information is lost. However it is possible to have a significant compression with little cost to accuracy performance, making this a worthwhile trade off.
An important result has shown that a large over-parametrised network pruned after training achieves higher performance than an equivalently sized network[**?**]. A larger network is more flexible, this higher degree of freedom allows for a better generalisation ability during training.
It is intuitively clear than not all weights in a model are equally important and contain the same amount of information. This is confirmed by the fact that random pruning of weights achieves worse results (measured as compression vs accuracy loss) than more guided methods. One way to select the best weights to prune is based on magnitude. This criterion can be applied layer wise or globally. It is possible to select a threshold under which we cut all weights or fix a target for level of compression. Layer wise pruning achieves worse results than global pruning, because in global pruning there is more flexibility since different layers can have different levels of sparsity. Regularization methods can be used for pruning, this approach has the advantage of combining training and pruning processes into one. Yet another method is to use an importance metric for the weights and eliminate the least important ones. An example is to evaluate the change in loss of the model after cutting some weight. A small change means that the weight is not very important for calculating the result of the model and we can therefore cut it.
All the mentioned methods apply the pruning techniques to the weights individually. This results in networks which do not have much structure and are sparse. Sparse matrix multiplications are not efficient to compute therefore other works [10] have applied pruning with coarser granularity, for example cutting groups of weights instead of individual ones. This leads to more structured connectivity and more efficient computations, meaning that even if we achieve a lesser degree of compression w.r.t. the previous methods, we can still evaluate the models faster.
Another intuition is that neurons in the network share the same information. Reducing the redundancy of information between neurons and layers is a good way to achieve good pruning trade-offs.[13] Worthy of mention is also the use of genetic algorithms to prune networks [35]. The optimization consists in finding the smallest subset of weights that provide the smallest loss of model performance.
All the mentioned pruning methods are applied after networks training. It has been shown, however, that it is possible to prune also before training. [33] The authors demonstrate that given a network and an initialization of it's weights it is possible to find a subset of the network which trained for the same amount of time on the same data achieves comparable performance to the starting model.

There is of course a limit to how much a network can be pruned. Cutting away too many neurons and connections can lead to a collapse of a layer and an unrecoverable loss of performance. This is due to the fact that too much pruning can degrade signal propagation in the network. [36]

2.8. Quantization

Quantization is the practice of transforming a continuous signal into a discrete value representation. The continuous signal has an infinite resolution, while the discrete value scale has a more limited representational power. In computer memory there cannot have true continuous values, so to record a signal it must be necessarily quantized. The goal of this transformation is to find the representation which minimizes the loss of information due to the lowered precision.
In the field of neural networks quantization refers to reducing the precision of the representation of weights, gradients, or activations in a network. The main purpose is the reduction of the memory occupation of the model, but it also results in faster computations. Computations on GPU are carried out using 32 bit FP numbers. It is possible to use lower bit FP numbers or integers, or to design entirely new number structures, such as a FP16 with 8 bit exponent instead of 5. The characteristics to consider are the range and resolution of available values.[26]
To quantize a value it is necessary to determine the range as well as a mapping between the original range and the target range of reduced precision. The mapping need not necessarily be uniform, it is also possible to have a logarithmic mapping or one determined in function of the specific values that need to be converted, by using a k-means algorithm for example. In the particular case of activation quantization for example the range of the values can be determined globally for the entire network or with a layer wise granularity. This opens up countless possibilities for designing quantization solutions and they have led to considerable network compression.
Quantization of a network can be applied to a trained network to optimize inference performance as well as during training. The reduction of number precision of a model reduces the degrees of freedom and flexibility. Quantization can induce a drift in the model behaviour, executing quantization iteratively in combination with training reduces this effect.[26] Quantization is a non differentiable transformation, which can be an issue when applied in training neural networks. In this case the straight through estimator can be used.
Quantization also leads to a speedup in model evaluation with the consequent reduction of inference time. In the domain of tinyML it is particularly advantageous since energy expenditure is reduced as well. Both these effects are due to the fact that operations on smaller number structures are more efficient and it also becomes easier to use vector instructions which have a higher throughput. Another very important consequence of using smaller number structures is that it enables support for more hardware.

2.9. Related work

The authors of Train++[31] present an incremental learning algorithm for binary classification in their work. Moreover, the solution is tiny since it works with only a few hundred kilobytes of RAM. It is fast and efficient, being able to train and perform inference in under 1 ms. However, it can only handle binary classification and relatively simple data with few features. It doesn't leverage the significant expressive power of neural networks. Nonetheless, a great advantage of this solution is that it is straightforward to set up for use.
The authors of tiny transfer learning[11] propose an on-device algorithm for neural network finetuning. Compared to Train++, this solution uses neural networks, which can solve far more complex problems. It is also not limited to binary classification. This method enables an order of magnitude reduction in training memory usage by freezing the weights and only training the biases. Bias updates do not require saving the intermediate network activations. However, freezing the weights reduces the flexibility of the model significantly. For this reason, the authors of TinyTL also introduce memory-efficient residual modules to maintain the model's adaptation capacity. This solution works on the raspberry pi 1 model A with 256 MB RAM.
The authors develop the TinyOL system[29] which acts as an extension that can be appended to a neural network deployed on-device. It is a dynamic layer that functions as a new output of the model and supports online training using gradient descent. This enables a degree of adaptation of the model in the field, making it possible to support concept drift or to incrementally learn new information. TinyOL is deployed on the arduino nano 33 BLE sense with 256 KB RAM.
The author of online on device transfer learning[16] develop a system for solving the problem of binary image classification. They employ the arduino nano 33 BLE sense with a camera module. The solution is made up of the mobilenet v1 model trained on the COCO dataset. It is deployed using TFLM and used for feature extraction. The extracted features are successively fed into a custom C implementation of a dense layer to produce a classification label. This approach is similar to our solution, with the main difference being that

our solution supports arbitrary feature extraction modules automatically. Furthermore the dense classification module is automatically generated and is not limited to only one layer. Lastly we have also the buffer for saving latent replays, which can be used to improve the quality of the model's adaptability.
The authors of latent replays[28] propose an original rehearsal strategy. The core idea is to save and replay some intermediate activation rather than input data. This allows a significant memory requirement reduction. In order to maintain the validity of the saved latent representations, it is necessary to freeze or slow down learning below the level at which this representation is produced. This solution enables effective continual learning, it is deployed to android smartphones through an app.

3. Solution description

At a high-level description, our solution is a system that takes as input a model and optionally some representative data and generates a solution to be deployed on-device with learning capabilities. We transfer the weights to maintain the weight initialization or the trained weights from the source model.

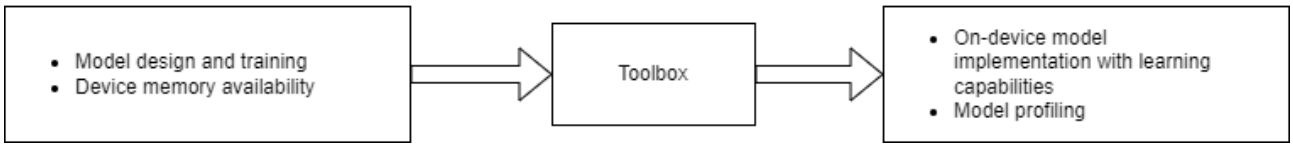


Figure 4: High level description of toolbox.

Let I be an input image with m rows, n columns and c channels, $I \in \mathbb{R}^{N \times M \times C}$ where $N, M, C \in \mathbb{N}$.
Let Φ denote a network such that it's architecture is composed of a feature extraction block Φ_f and a classification block Φ_c and y is its output, so $y = \Phi(I)$. The input is processed by Φ_f and a feature vector of size $|\psi_I|$ is extracted from I. The feature vector ψ_I is the input of Φ_c which produces the classification label $y = \Phi_c(\psi_I)$ and $\psi_I = \Phi_f(I)$.
It is also possible to work with models without a feature extraction module Φ_f. In this case $\Phi \sim \Phi_c$ and the input I is fed directly into Φ_c, therefore the output label is $y = \Phi_c(I)$.
There are no constraints on Φ_f besides those imposed by the specific tool for on-device implementation, which in our case is TFLM. In this work Φ_f is a convolutional neural network with two convolution blocks composed of a convolution layer followed by a maxpool layer. Φ_c is a dense feedforward network with one or more layers. Each dense layer has an activation function, the activations supported by the current implementation version are sigmoid and softmax.
The network has k layers L, one of which is a flatten layer $L_{flatten}$ separating Φ_f and Φ_c. Given a layer L_i, θ_i denotes its weights. $\theta_{i,j}$ indicates the weights of neuron j of layer i and $|\theta_{i,j}|$ is the cardinality of the set of weights of neuron j, in a fully connected network without skipped connections it is also the number of neurons in layer L_{i-1}.
In the case of convolutional layers, there are f filters with r rows, s columns and t channels, $L_{conv} \in \mathbb{R}^{f \times r \times s \times t}$. S_i will denote the stride along dimension i.
For average pool or max pool layers, r, s and S_i denote the number of rows, columns, and stride respectively.

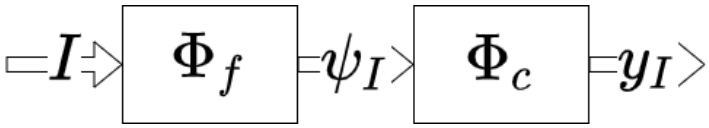


Figure 5: Schema of the stages of input processing.

3.1. Profiling

Once a model is ready, the profiler provides a measure of the memory and computational requirements. For every layer L_i, the profiler computes the number of parameters p, the size of the activations mem_{act}, and the number of operations required to evaluate the inputs. The latter is expressed as the number of flop n_{flop}, macc n_{macc}, division n_{div}, sum n_{sum}, and comparison operations n_{cmp}. The profiler is a tool to aid the neural networks' design that will be deployed on-device. In particular, the estimation of the activation size is helpful to discard designs that do not satisfy the device memory constraints.
In the following tables, we present the computation of the above values.

Type of layer	Number of parameters
Conv2D	$(r \cdot s \cdot t + 1) \cdot f$
Dense	$\lvert\theta_{i,j}\rvert \cdot (\lvert\theta_{i-1,j}\rvert + 1)$

Table 1: The number of parameters for different types of layers.

Type of layer	MACC	FLOPs
Conv2D	$(\frac{m-r}{S_y}+1)(\frac{n-t}{S_x}+1)(f\cdot(r\cdot s\cdot t+1))$	$(\frac{m-r}{S_y}+1)(\frac{n-t}{S_x}+1)(f\cdot(2\cdot(r\cdot s\cdot t+1)))$
Dense	$m_i\cdot n_i\cdot c_i\cdot(m_{i-1}\cdot n_{i-1}\cdot c_{i-1}+1)$	$2\cdot m_{i-1}\cdot n_{i-1}\cdot c_{i-1}\cdot m_i\cdot n_i\cdot c_i$
Depthwise conv2D	$(\frac{m-r}{S_y}+1)(\frac{n-t}{S_x}+1)(r\cdot s+1)\cdot t\cdot f$	$(\frac{m-r}{S_y}+1)(\frac{n-t}{S_x}+1)\cdot 2\cdot(r\cdot s+1)\cdot t\cdot f$

Table 2: The number of FLOP and MACC operations for different types of layers.

Not all layers, however, are computed with MACC operations. FLOPs, on the other hand, are not an ideal measure of computational load since it is hardware dependent. Therefore, we measured the computational load in the number of operations used for the specific layers in a baseline implementation for better clarity. Despite this, FLOPs provide a single value estimating the complexity which is useful for making simpler comparisons between different models.

Type of layer	Sums	Divisions	Comparisons	FLOPs
ReLU	0	0	$m\cdot n\cdot c$	$m\cdot n\cdot c$
Max pooling 2D	0	0	$m\cdot n\cdot c\cdot r\cdot s$	$m\cdot n\cdot c\cdot r\cdot s$
Average pooling 2D	$m\cdot n\cdot c\cdot r\cdot s$	$m\cdot n\cdot c$	0	$m\cdot n\cdot c\cdot(r\cdot s+1)$
Add merge	$m\cdot n\cdot c$	0	0	$m\cdot n\cdot c$

Table 3: The number of operations for different types of layers.

The size of the activation of a certain layers is the size of the output activation map times the size of the datatype used to represent values. The output activation map depends on the layer: for a dense layer it is the number of neurons, while for a convolutional layer it is the number of filters times the size of each channel which depends on the size of the input, the dimension of the convolutions and the stride employed.
$Size_{datatype}$ denotes the size of the datatype used to store values. In this work the activations and weights are stored as 32 bit floating point values.

Type of layer	Activation size
Conv2D	$(\frac{m-r}{S_x}+1)\cdot(\frac{n-s}{S_y}+1)\cdot Size_{datatype}$
Dense	$\lvert\theta\rvert\cdot Size_{datatype}$
Avg pool, max pool	$m/r\cdot n/s\cdot Size_{datatype}$
ReLU	$m\cdot Size_{datatype}$

Table 4: Activation size for convolution and dense layers

3.2. Model conversion

The first step in creating an on-device implementation is analyzing the model and extracting all the information required for code generation. These include the layers' structure, dimensions, and weights for generating the code to implement the model. We extract these parameters from the input TensorFlow model provided to the toolbox.
For on-device deployment, we split the model into the two components described above, Φ_f and Φ_c. This step is done automatically and is guided by the presence of the flatten layer $L_{flatten}$ separating Φ_f and Φ_c. Of the two, Φ_c is the most relevant in this work since it will have learning capabilities. Φ_f, on the other hand, can be deployed using TensorFlow lite micro (TFLM) or other equivalent tools. For our experiments, we have used TFLM.

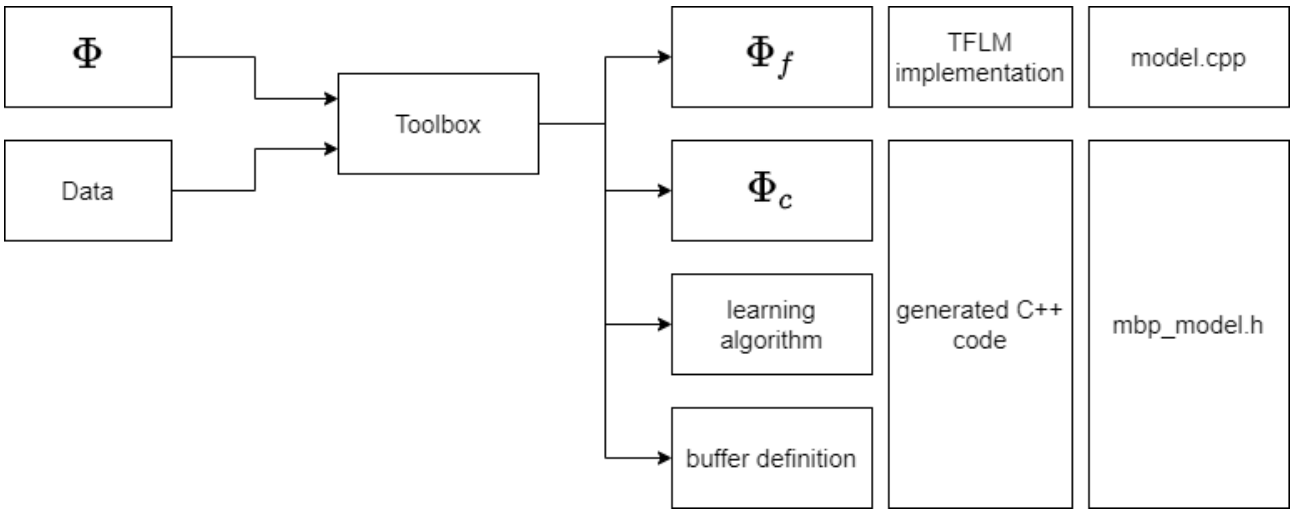


Figure 6: Toolbox description.

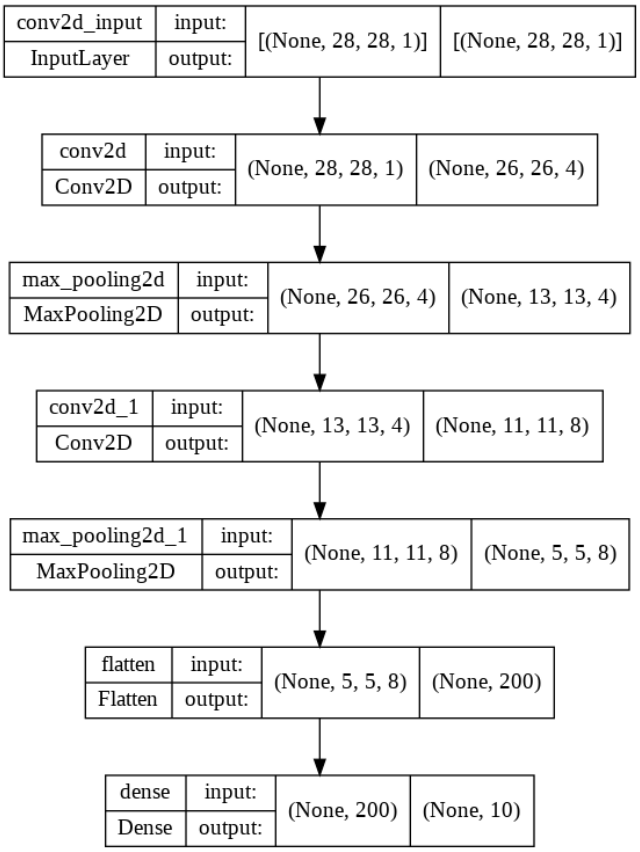
In order to be able to create prototypes and execute experiments faster and without the limits of on-device hardware, the toolbox also contains a python implementation of Φ_c which is equivalent to the C++ implementation that is deployed on-device. The python implementation will be used for more extensive experimentation and the results will be validated by some on-device reproductions.
The information extracted from model analysis are the input to the code generation part of the toolbox. The output is a .h C++ file which contains the model implementation as well as the implementation of all the algorithms for model evaluation and training.
Note that the solution is modular, it can also be used without a feature extraction part. Cutting away the feature extraction block limits the complexity of the data that solutions can be applied to, but this shows that the two blocks are truly independent. We perform an experiment and a deployment also without feature extraction (described in chapter 5.2).

3.3. Buffer

After calculating the memory requirements of the network, knowing the total available memory on-device we allocate a buffer B where we can store feature samples.
Given that this solution only trains the classification head of the model deployed with the custom implementation, there is no need to store the raw data samples. We only store the features ψ_I, the inputs to the first layer of the block to be trained Φ_c. This allows significant reduction in the memory requirements for storing samples which would otherwise be prohibitive for resource constrained devices.
At runtime there is a stream of input data I, for our purposes it is infinite. No assumptions can be made on the order in which the data arrive. Most samples will be unlabeled, for those the network will provide a calculated target $y = \Phi(I)$. Some data we assume will be labelled. This labelled data will be used to adapt the model. The buffer acts like a sliding window over the input stream of labeled data. Having the ability to save feature samples in memory greatly boots learning performance since this allows to perform multiple training passes over the data.

3.4. Implementation details

The toolbox is implemented in python and it has three components. Firstly the profiler and model analysis tool. It operates on the provided model to measure the computational requirements of the different layers and the memory occupied by the activations, which take the largest share of RAM. Information such as model structure and size of layers is extracted. These will be needed for generating the C++ implementation. The weights from the original model are extracted as well in order to be transferred on-device.



(a) Structure of model being profiled

```
0 {'result': (18928.0, 27040.0, 10816.0), 'parameters': 40, 'operations': {'flops': 18928.0, 'maccs': 27040.0, 'divisions': 0, 'sums': 0, 'comparisons': 0}}
1 {'result': (2704, 2704.0), 'parameters': 0, 'operations': {'flops': 0, 'maccs': 0, 'divisions': 0, 'sums': 0, 'comparisons': 2704}}
2 {'result': (96800.0, 38720.0, 3872.0), 'parameters': 296, 'operations': {'flops': 96800.0, 'maccs': 38720.0, 'divisions': 0, 'sums': 0, 'comparisons': 0}}
3 {'result': (800, 800.0), 'parameters': 0, 'operations': {'flops': 0, 'maccs': 0, 'divisions': 0, 'sums': 0, 'comparisons': 800}}
4 None
5 {'result': (4000, 2010, 40.0), 'parameters': 2010, 'operations': {'flops': 4000, 'maccs': 2010, 'divisions': 0, 'sums': 0, 'comparisons': 0}}
total parameters:  2346
total activation size:  18232.0
max activation size 10816.0
total flops:  119728.0
total maccs:  67770.0
total divisions:  0
total sums:  0
total comparisons:  3504
```

(b) Model profiling output

The second component is used for generating C++ code to be deployed on-device. Consider the following small example:

```
def gen_network_struct(number_of_layers):
    res = "typedef struct t_dense_network{{\n\
        float * bias_list[{}];\n\
        float * weight_list[{}];\n\
        float * layer_list[{}];\n\
    }}t_dense_network;\n\
    t_dense_network dense_network;\n"
    .format(number_of_layers - 1, number_of_layers - 1, number_of_layers)
    return res
```

We have implemented the model to be deployed on-device using standard C++ data structures which make it possible to implement the learning algorithm as well. It is not possible to implement a learning algorithm on top of the TFLM model implementation. This is due to the fact the the TFLM modle is implemented using flatbuffers[4] which is an unmodifiable serialized data structure. The flatbuffer allows read access to data without deserialization, it is therefore very fast and memory efficient. Modifiying weights in the training procedure, on the other hand, would require to deserialize and create a new flatbuffer at every weight update, which is unfeasible.
The last component is a python implementation equivalent to the C++ implementation generated for on-device

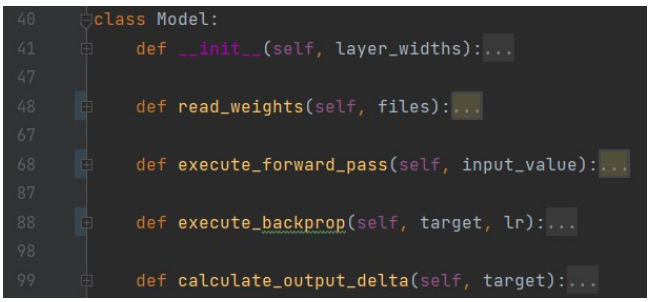


Figure 7: Snippet of python implementation.

deployment. The rationale behind this functionality is to allow for the possibility to execute experiments unconstrained by on-device hardware limits. It is also helpful in prototyping and testing.

4. Algorithms

4.1. Buffer management

The fundamental problem that the buffer needs to solve is the mapping between the infinite stream of data that the device records and a limited storage from which we can access data to update a model.
The simplest solution is to fill the buffer with the data in the order that they arrive until we run out of space. Then perform training and flush the buffer.
An alternative solution is to use the buffer as a FIFO queue. In this case a new sample is stored in the buffer if there is space, otherwise it replaces the oldest sample.

Algorithm 1 Batch buffer management

```
1: Empty buffer of size B
2: while Buffer is not full do
3:     Receive data sample
4:     Calculate features
5:     if Data is labelled then
6:         Add feature vector and label to buffer
7:     end if
8: end while
9: Train
10: Flush buffer, goto 1
```

Algorithm 2 FIFO buffer management

```
1: Empty buffer of size B
2: while True do
3:     Receive data sample
4:     Calculate features
5:     if Data is labelled then
6:         if Buffer not full then
7:             Add feature vector and label to
               buffer
8:         else
9:             Replace oldest sample
10:        end if
11:    end if
12:    Train
13: end while
```

It is possible to adopt some considerations in order to make better use of the limited space buffer. Based on the technique of latent replays [28], save in the buffer an intermediate feature vector instead of full input samples. The label associated to a sample is also stored in the buffer alongside the features.
An important consideration is on the determination of the size of the buffer. It is constrained by the size occupied by the network parameters and the activations of the classification module. The profiler provides a measurement of these values, subtracting these values from the total memory availability gives us the buffer size we can afford. Below is an example in the particular case of the network for MNIST digit classification which has been used in the transfer learning experiment described in section 5.3.
M denotes the total memory available on-device. mem_{Φ_f} and mem_{Φ_c} denote the memory used by the Φ_f and Φ_c modules. mem_{Φ_f} includes the memory occupied by the TensorFlow lite micro interpreter.

$$B = M - mem_{\Phi_f} - mem_{\Phi_c} = 256KB - 70KB - 10KB = 176KB$$

We have $176KB$ left for the buffer. The size of the feature vector ψ_I extracted by Φ is 200 and each feature is a $32bit$ floating point number. With this information we can calculate the number of samples that the buffer can store:

$$N_{samples} = \frac{B}{Size_{datatype} \cdot |\psi_I|} = \frac{176KB}{32bit \cdot 200} = 220$$

We can have a buffer of up to 220 elements for this particular setup.

4.2. Code generation

To run a model on-device, we need a definition of the model and the implementation of all the methods necessary to perform the desired functionalities. Furthermore, we target not just a single problem but create a general tool for training feedforward neural networks on tiny devices. Consequently, the solution must support models of different shapes and sizes automatically.
The solution is a two-step process: first, extract from the input model all the information defining the model. Second, use this information to generate a suitable model definition for running on the device. At a conceptual level, we are performing a translation. More concretely, our python program inspects the TensorFlow model and extracts structure, size, and weights in the first step. This information is used to generate an equivalent model definition using C++ data structures, which can be used on microcontrollers with C++ support.

Memory is one of the tightest constraints in the domain of tinyML. There is also no garbage collection or automatic memory management. For this reason, it is crucial to avoid the use of dynamic memory allocation, which over time would lead to memory fragmentation and inefficient use of the resources. Dynamic memory allocation also prevents us from precisely calculating the program memory usage before runtime, which can lead to unexpected crashes. Our solution does not employ dynamic memory allocation, memory use can be calculated at compile time.

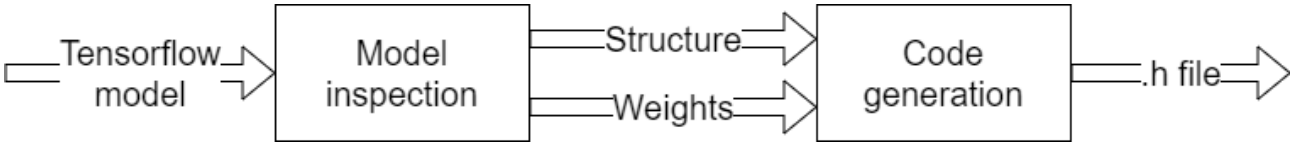


Figure 8: Schema of code generation steps.

Once this process is complete the generated file can be imported into the arduino project and the hello_world_learn file contains the application that uses the model.

4.3. Backpropagation

We have implemented the learning procedure using backpropagation of errors[30]. To execute backpropagation we use the chain rule of derivatives to propagate the loss derivative from the output of layer l_i to that of layer l_{i-1}. Back propagation is the generalization of the delta learning rule to multi layer FF networks: the learning signal is

$$r = [d_i - f(w_i^T x)] * f'(w_i^T x)$$

where d_i is the supervision feedback and f is a function of the activation and r is the learning signal. This is the perceptron learning rule applied to continuous functions. The weight update consequently is

$$\delta w_i = \alpha(d_i - o_i) f'(w_i^T x) x$$

where α is the learning rate, o_i is the output activation, x is the input.

5. Experiments

5.1. Validating toolbox implementation

The first test is done to verify that the model implemented with the toolbox is equivalent to the original TensorFlow implementation. To do this we start with a model and we translate it with the toolbox. Then we compare the activations, which we expect to be the same, proving that the weight transfer works properly and the model is implemented correctly. Below is scheme of the model used and the comparison of the activations in the output layer.

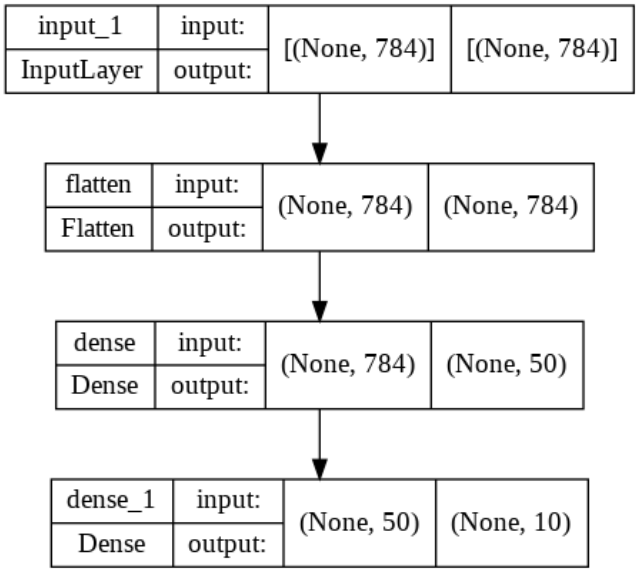


Figure 9: Model used for toolbox implementation validation

TensorFlow	toolbox
0.49128962	0.4912895765018331
0.35619456	0.356194561415557
0.6033897	0.6033897061250977
0.32689995	0.3268999920168154
0.2315377	0.23153772458133343
0.77069306	0.7706930758837134
0.3988884	0.3988884301820282
0.6468067	0.6468066535447842
0.26513988	0.2651398574900999
0.73972404	0.7397239561245914

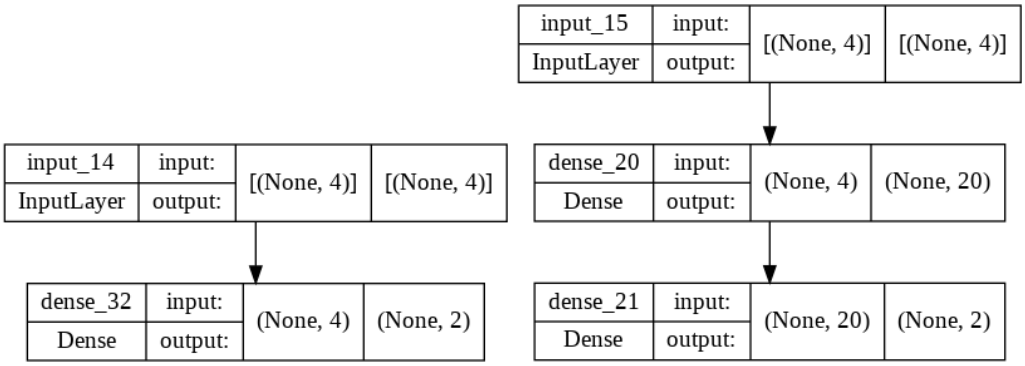
Table 5: Activation comparison between different implementations.

The activation values are within 10^{-7} between the different implementation. This difference is small enough to be attributed to floating point number uncertainties. With this result we can confidently state that the toolbox implementation is comparable to the TensorFlow implementation in terms of model behaviour correctness.

5.2. Validation of learning capabilities

To verify that this solution is able to train a network we convert an untrained network with the toolbox and execute the learning procedure.
Since the toolbox enables training only of the dense feedforward block of the network we will focus on banknote authentication dataset for this part, without convolutional feature extraction.

The task is binary classification: the model has to learn to distinguish between authentic and fake banknotes based on features extracted from the scans.
The first step of this procedure is to design a suitable network for the target dataset.

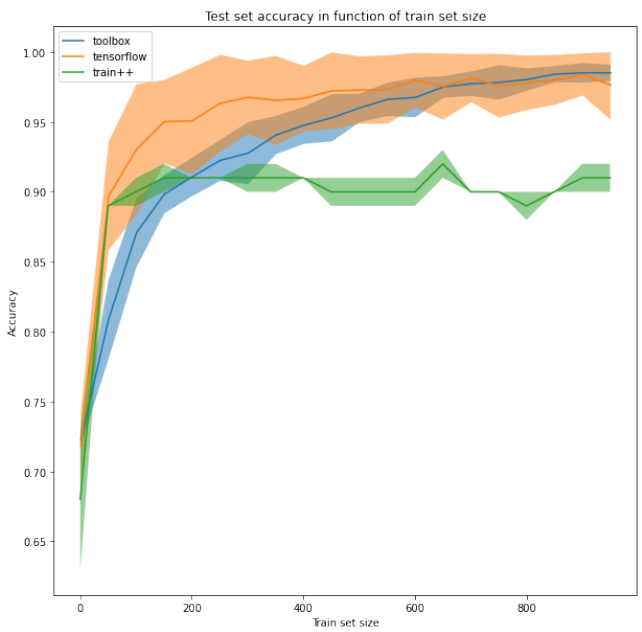


(a) Model 1 (b) Model 2

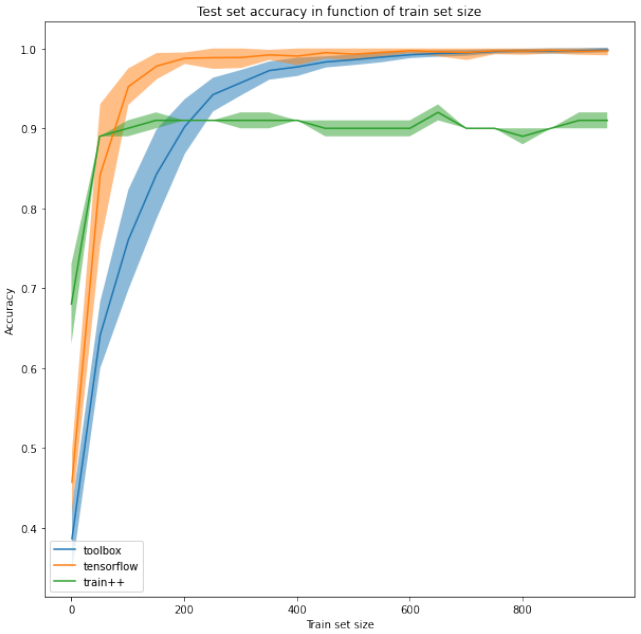
The models used in this experiment are composed of just the Φ_c module with the learning capability.
We test a model with just one layer with softmax activation as well as one with a hidden layer with sigmoid activation to show that this system can support hidden layers.
Once we have designed a network for the dataset we convert it using the toolbox.

```
m_python, mc_header = toolbox.generate_implementation(model)
```

Here m_python is the python clone of the on-device implementation and mc_header is a string containing the on-device implementation which is written to file and added to the arduino project. This function is designed to convert models composed of just a Φ_c module, without feature extraction Φ_f. Therefore it also doesn't require a flatten layer.
We will also be comparing learning performance with the Train++[31] solution.
The network is trained for 1 epoch on a dataset of increasing size to evaluate the accuracy achieved in function of the training set size. This is the case for both the toolbox model as well as the comparison tensorflow model. The TensorFlow model used for comparison has exactly the same architecture as the toolbox model, in fact the toolbox model is simply an alternative implementation that is suitable to run on-device.
The TensorFlow model is compiled with accuracy as the metric, SGD as the optimizer, and sparse categorical crossentropy for calculating the loss. Data are seen one at a time and discarded afterwards in the manner of online learning. The learning rate is fixed and equal between the model in our implementation and the TensorFlow implemented model. The value is determined with a small validation set disjoint from the train set and the test set.



(c) Model 1 performance comparison



(d) Model 2 performance comparison

This graph represents the accuracy achieved on the test set of the banknote authentication dataset in function of the size of the training set by three different solutions. TensorFlow works best in terms of final accuracy achieved, and we will use it as an upper bound comparison. However, TensorFlow is not a tiny solution, and the TFLM framework does not support training.
Train++ [31] is a tiny and very efficient solution. However, it only supports binary classification and cannot handle complex problems. It also achieves a lower final accuracy.
Our solution is tiny and supports dense feedforward neural networks. It also achieves a final accuracy score on-device comparable to TensorFlow's.

5.3. Evaluation for transfer learning

One application of the ability to train neural networks on-device is transfer learning. In this setting there is a source domain with dataset D and a target domain with a dataset $\bar{D}$ such that $|\bar{D}| << |D|$. If the source and target domain share common features or the features of the source domain are meaningful in the target domain, it is possible to transfer the training on D to achieve better performance on the target task than what would be achieved by training just on $\bar{D}$.
Transfer learning is achieved by adapting Φ trained on D. The weights of the feature extraction layers (Φ_f) are frozen, and the layers close to the output Φ_c are trained on $\bar{D}$. Freezing the weights allows the model to use the features of the source domain to learn the target task.
The MNIST dataset is composed of 28×28 size grayscale images representing single digits. The label of each image is the corresponding integer. The task is to identify the digit in the image.
In this particular experiment, we train a model to recognize the even MNIST digits, then execute transfer learning on the task of recognizing odd digits. More precisely, D is the subset of the MNIST dataset with even digits, and $\bar{D}$ is the subset with the odd digits.
The model Φ used for this experiment has the Φ_f module composed of 2 convolutional blocks, each of which is made up of one convolutional layer and a maxpool layer. The features are unrolled at the flatten layer and passed as input to the Φ_c module composed of a dense layer with softmax activation.

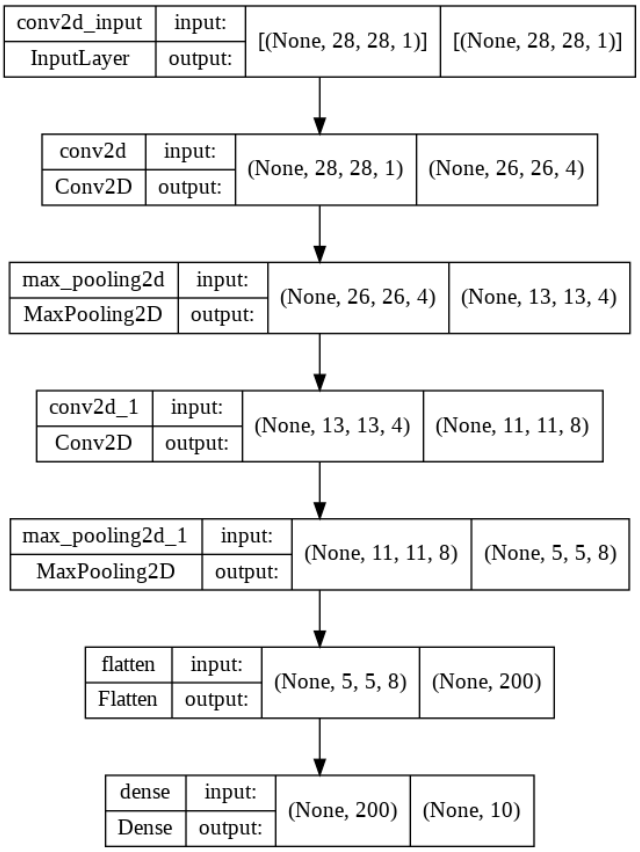


Figure 10: Model used for transfer learning experiment.

Calculating the affordable buffer size as described in section 4 gives a feasible buffer size of 230 for this particular model configuration which produces a feature vector size $|\psi_I| = 200$. The buffer is used as a FIFO queue as described in algorithm 2 of section 4.
The TensorFlow model used for comparison is compiled with SGD as optimizer, the loss function is sparse categorical crossentropy. Each training pass is done for 1 epoch and the batch size is 1.

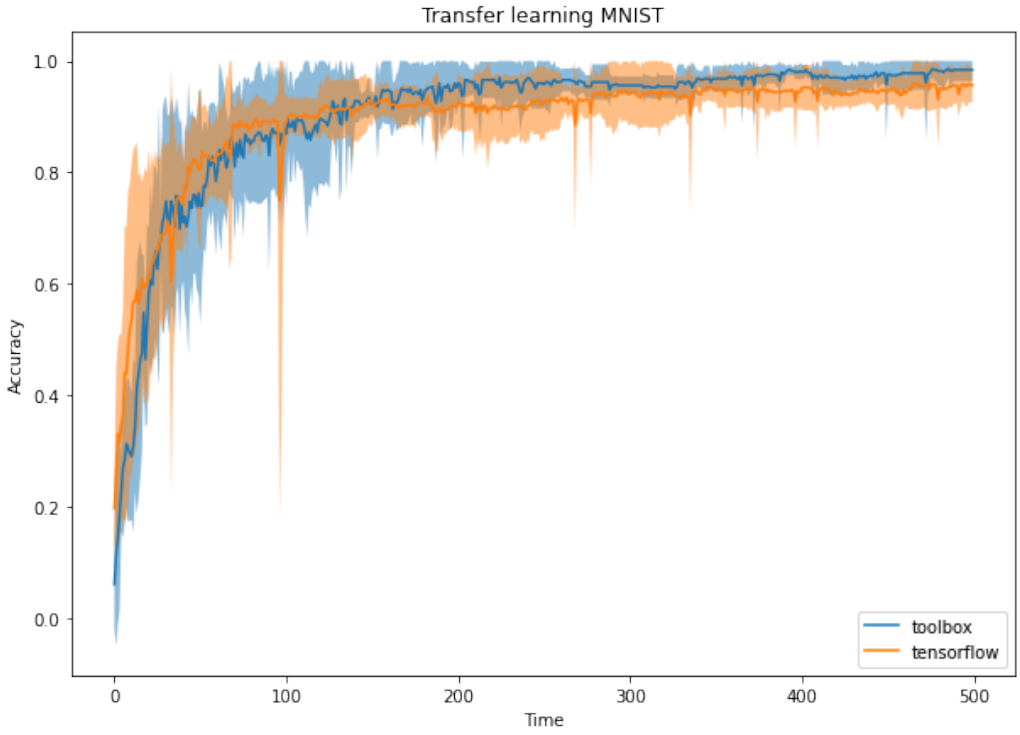


Figure 11: Transfer learning performance: even digits to odd digits

At every time point, the model receives a labeled sample I. In all experiments, a time step is defined by the arrival of a new labeled data sample in input. Then, the associated feature vector ψ_I is calculated. The training of both the TensorFlow model and the toolbox model make use of the FIFO buffer management strategy described is chapter 4. If there is space in the buffer, this sample is saved. Otherwise, it goes to replace the oldest sample in the buffer. Afterward, the model performs a training pass over all the saved feature vectors in the buffer.
The toolbox can fine-tune Φ_c so that the model Φ learns the target task using the feature extraction module Φ_f trained on the source domain dataset.
We perform a second transfer learning experiment from the same model trained on the full MNIST dataset to the task of recognizing fashion MNIST objects. More precisely, D is MNIST while $\bar{D}$ is fashion MNIST.

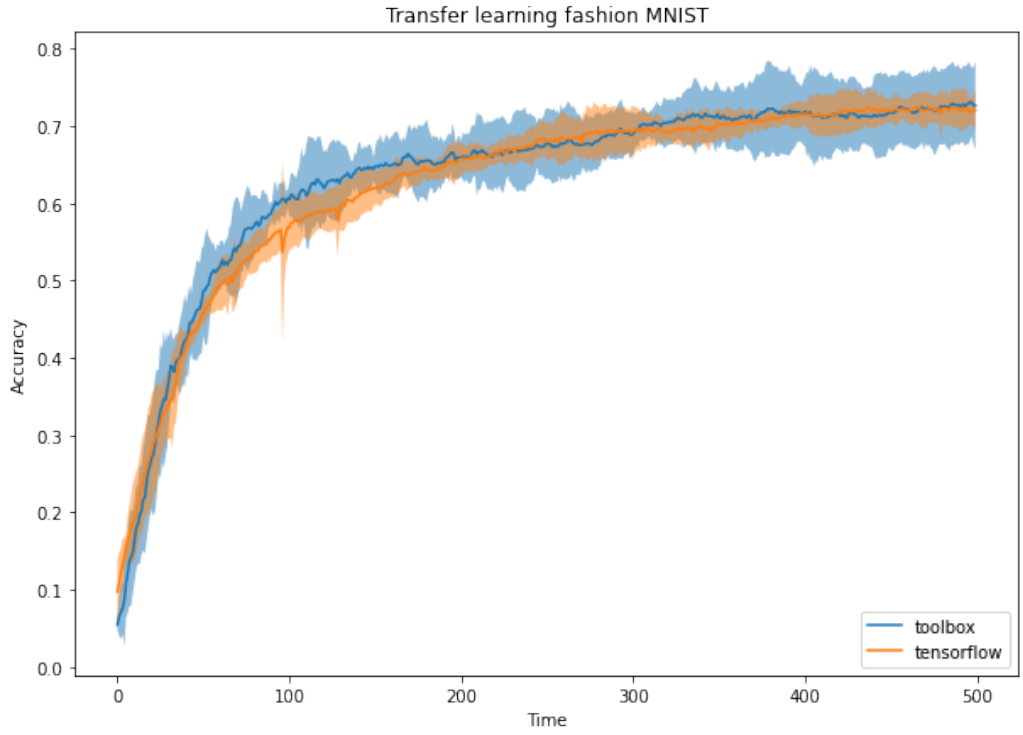


Figure 12: Transfer learning performance: MNIST to fashion MNIST

This confirms the transfer learning capability for a more complex problem. We can adapt a network directly on-device to a new target problem where the features from the source task are transferable. Acceptable performance is learned and convergence is achieved with a reasonable amount of data.

5.4. Evaluation in concept drift scenario

We can adapt a neural network on tiny devices, which can be helpful to make a model more resilient in concept drift scenarios.
The problem of concept drift is defined by a change in the task solved by the model. The change can be an evolution of the data distribution or a change in the target class definitions. Furthermore, the change can be gradual or abrupt.
We are working with the MNIST dataset in an abrupt concept drift scenario for this experiment. We have the standard data D, and the dataset with a concept drift $\overline{D}$, which we produce by executing a class swap between digits 6 and 4.
The model architecture employed is the same as described in section 5.3.
Following is the experiment setup: we design and train a model on the MNIST dataset D. Then we convert the trained model using our toolbox and evaluate the model.

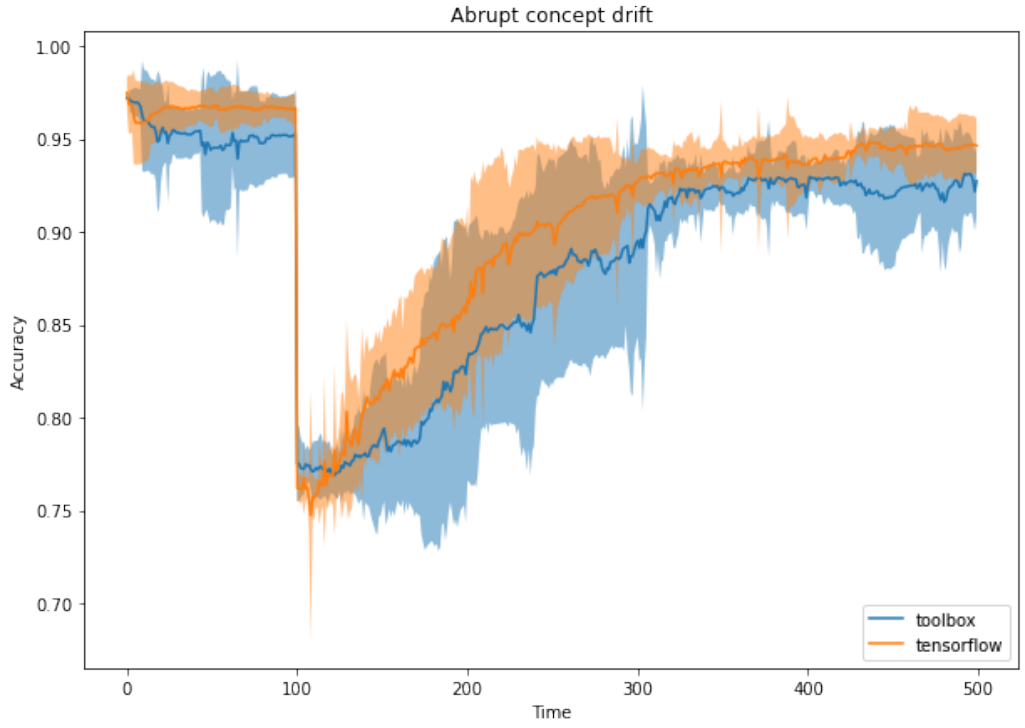


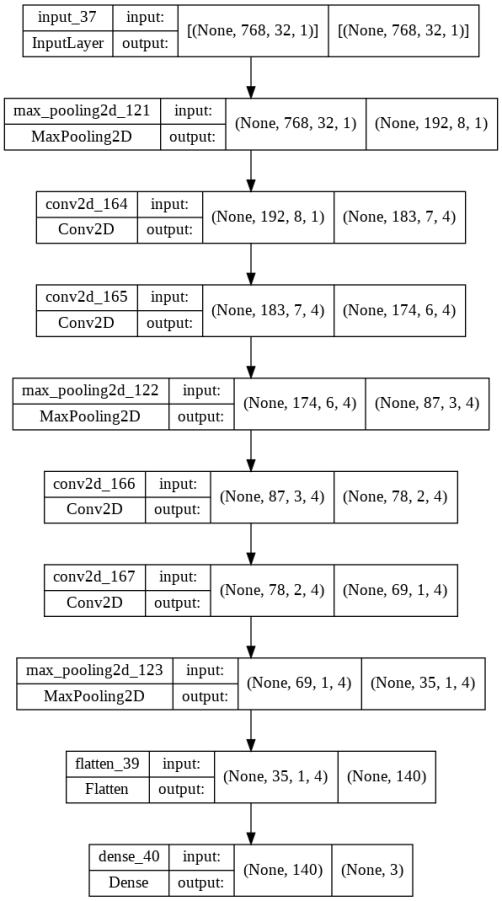
Figure 13: Model accuracy trained after concept drift.

The abrupt concept drift event occurs at time t=100. Each time step is the arrival of a new labeled data sample. The buffer size is 100 and the buffer operates as a FIFO queue as described in algorithm 2 of chapter 4. Consequently, when concept drift occurs, the buffer is full of data representing the old task. There is a drop in performance from 0.95 accuracy to 0.74 due to the abrupt concept drift. The model uses the new data to adapt. It takes some time to recover performance because the buffer stores old data, which are gradually replaced. The time to recover from the concept drift depends on the buffer's size and the difficulty of learning the new task. This shows that our solution enables a model deployed on-device to adapt after an abrupt concept drift. This eliminates the need to train a new model and redo the entire deployment process in the event of a concept drift.

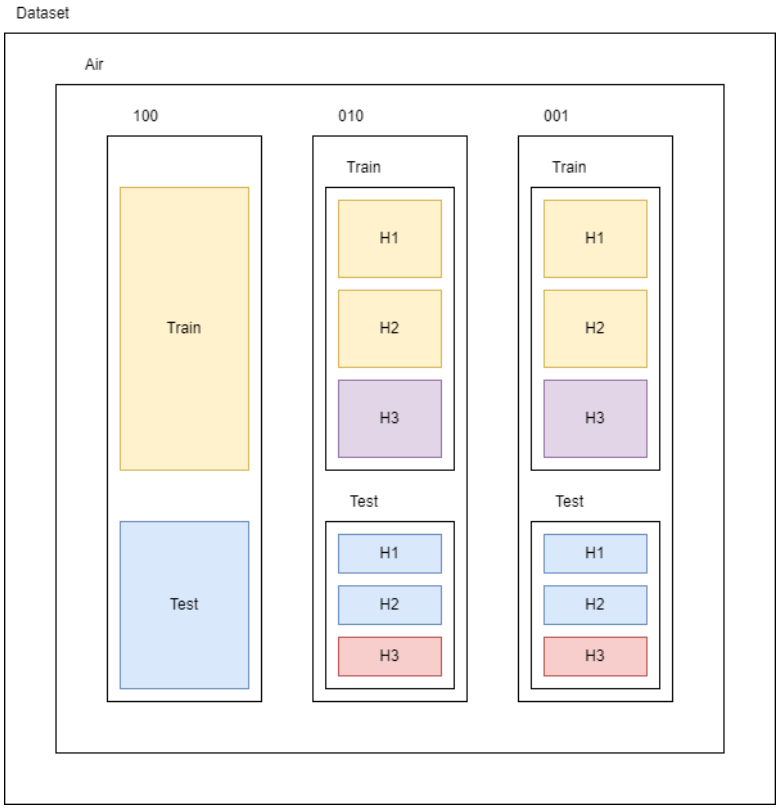
5.5. Incremental learning

The goal of this experiment is to verify one of the applications of this solution, more precisely, incremental learning.
We will also be using a new dataset: IR-UWB radar measurements of human activity[5]. The radar works by emitting an impulse and measuring the reflected energy from the environment. The data provided comprises $768 \times 32 \times 1$ size matrices in which measured energy is recorded at different points in space and time. The radar records data in a room where one of three available volunteers can either be standing or walking or the room can be empty.
The task is to detect the person and distinguish between standing and walking.
The incremental learning problem is defined as follows: there is a network Φ trained on task T over dataset D. The task is to detect and classify human activity from an input consisting of IR-UWB radar data measurements. There are three possible classes corresponding to an empty room, a person standing or walking. There are recordings of three different persons. In the figure below schema (b) describes the structure of the dataset.
The neural network used in this experiment, described also in schema (a) below, is different from the one used in the other experiments. However, it follows the same architecture: a feature extraction module Φ_f composed of convolution and maxpool layers followed by a classification module Φ_c made up of a dense layer with softmax activation.
The equivalent TensorFlow model used for comparison is compiled with SGD optimizer and sparse categorical crossentropy loss, equivalent to all previous experiments.
The model Φ is trained on data from just two subjects. At this point, Φ achieves a great accuracy in detecting persons 1 and 2 on which it was trained. This performance translates to data gathered on person 3, but with a moderate loss.
The objective is to learn to detect person 3, while avoiding forgetting previously learned knowledge. The result is evaluated by measuring the model's performance on the old and new tasks. The one described is a concrete

example of continuous learning, as described in chapter 2.



(a) Schema of the model used in the incremental learning experiment



(b) Schema of the IR-UWB dataset partition

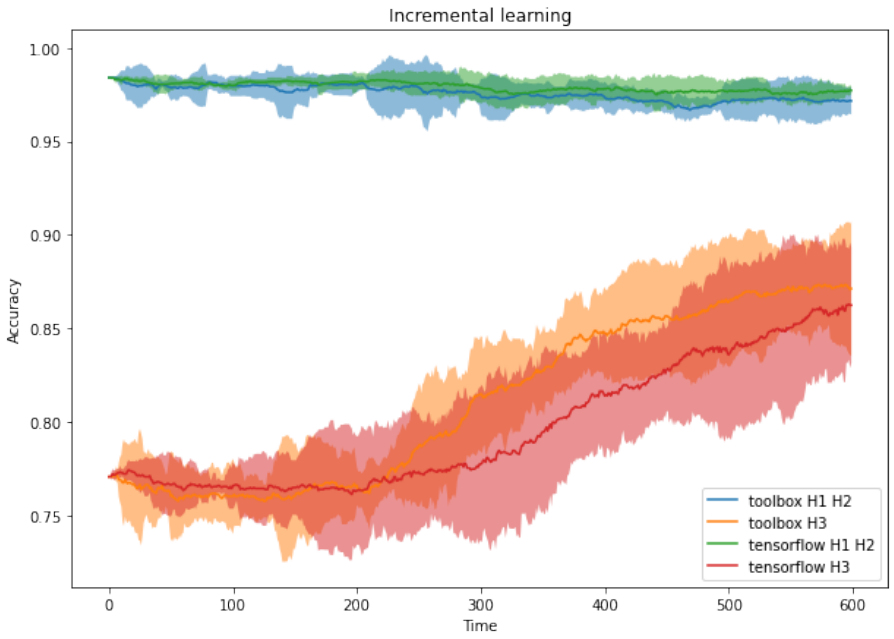


Figure 14: Accuracy w.r.t. train set size in incremental learning scenario

In this experiment the buffer size is 200 and it operates as per algorithm 2 of chapter 4. The new subject is introduced at t = 200.
Here we see a noticeable improvement in performance on the new task, while the loss on the first task is tolerable. This proves the ability to continually learn a task and expand its scope directly on-device, enabling applications of incremental learning and algorithm personalization.

6. On-device deployment

We have tested the baseline Train++[31] and our solution on the Arduino nano 33 BLE sense[1]. It has an ARM Cortex M4 MCU running at 64MHz clock and 256 KB SRAM.

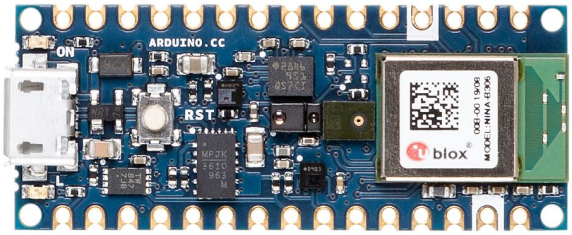


Figure 15: Arduino nano 33 BLE sense.

The TinyML team at google has produced example projects[8] which we used as a starting base for development. The code deployed on-device includes the model specification generated by our toolbox and the C++ code implementing the learning functionalities.
The model.h and model.cpp files contain the TFLM implementation of Φ_f. The mbp_model.h file contains the implementation of Φ_c. The TLFM interpreter must be set up in order to run Φ_f.
The firmware flashed on the device reads the serial port over which we send data or control characters. The control characters can be used to instruct the device to perform training or to write the current accuracy scores back over the same serial port. The data received is appropriately interpreted and passed as input to the model which is then invoked. If a label is also present we save the features extracted from the data sample. We also compare the model prediction to the label and update the accuracy score.
The data is streamed to the device from a file by the serial_server.py script. The format of the data is the following: each row contains a sample with values separated by commas. The target label is optional and is the last element of the row. The control characters are 't' for train and 'r' for report. The code for generating the files of data properly formatted to be streamed to the device from numpy arrays is also provided.
To execute an experiment we flash the firmware to the device, then execute the serial_server.py program which streams a training set and a test set to the device and then reads the accuracy achieved by the model on the test set after one epoch of training on the train set.

6.1. Nonfunctional characteristics

We measure the speed of execution of inference and training of the model used for the transfer learning and concept drift experiments described in section 5.3.
The measurements are done using a $100MHz$ oscilloscope. The program measured runs on the arduino nano 33 BLE sense, one $3.3V$ pin is connected to one of the oscilloscope channels. Below is the required setup:

```
\\ preprocessor directive:
#define OUT 2
\\ in setup:
pinMode(OUT, OUTPUT);
\\ in loop:
digitalWrite(LED_BUILTIN, HIGH);
\\ {the code to be measured}
digitalWrite(LED_BUILTIN, LOW);
```

Other measurements are done using software timing tools from Arduino libraries using the following setup:

```
unsigned long time_invoke = 0;
int count_passes = 0;
unsigned long start = millis();
TfLiteStatus invoke_status = interpreter->Invoke();
unsigned long end = millis();
time_invoke = (time_invoke * count_passes + (end - starti)) / (count_passes + 1);
```

```
count_passes++;
```

The two methods render comparable results. In the following table we report execution times for the model described in section 5.3.

Module	Forward pass	Backpropagation
Φ_c	625μ s	$682\mu s$
Φ_f	$63ms$	–
$\Phi(TFLM)$	$64ms$	–

Table 6: Execution time measurements with software tools.

Consider also the execution times of the model described in section 5.2, for which we can also make a comparison with Train++:

Module	Forward pass	Backpropagation	Total
Φ_c	86μ s	$5\mu s$	$91\mu s$
$Train++$	–	–	$39\mu s$

Table 7: Execution time measurements with software tools.

7. Conclusions

In this thesis we have shown that it is possible to have adaptable neural networks in the domain of TinyML. At time of delivery this is also the first solution to generate code for implementing a neural network on-device with learning capabilities.
We have also worked on measuring the memory requirements of the activations and computational requirements of the layers, building the profiler which shortens the design phase and eliminates the need for trial and error in deploying models.
Our experiments cover training from scratch, transfer learning, concept drift and incremental learning scenarios with publicly available datasets. This solution does not focus on one network or one problem, but is a framework with a wider scope.
Measurements have also shown that our solution is able to execute the adaptation algorithm in under 1ms per sample with feature vectors of 200 elements.
An important principle has been reproducibility of the work, all code is open source, in particular also the code for the experiments and auxiliary tools.[27]

7.1. Future work

Future work includes expanding the implementation of these tools by adding batch training to make use of multiple CPU cores when they are available as well as expanding support for more learning algorithms, activation functions, and layers.
For a further improvement of the buffer consider that neural network training algorithms work best on independent and identically distributed ($i.i.d$) data, in fact, when using non $i.i.d$ data the phenomenon of catastrophic forgetting is prominent. This phenomenon is present not only when training the network on an extension of a task or with new data, but also between single batches of data[22]. We can use the available buffer to collect data from the input stream so as to optimise training and avoid as best as possible catastrophic degradation in performance. This can be done by accumulating in the buffer a balanced set of data.

8. Bibliography and citations

References

[1] Arduino nano 33 ble sense.

[2] Banknote authentication dataset.

[3] Fashion mnist dataset.

[4] Flatbuffers white paper.

[5] Ir uwb through wall radar human motion status dataset.

[6] Mnist dataset.

[7] A proposal for the dartmouth summer research project on artificial intelligence.

[8] Tensorflow lite micro examples.

[9] Tinyml for uwb-radar based presence detection.

[10] Sajid Anwar and Wonyong Sung. Compact deep convolutional neural networks with coarse pruning. *arXiv preprint arXiv:1610.09639*, 2016.

[11] Han Cai, Chuang Gan, Ligeng Zhu, and Song Han. Tinytl: Reduce memory, not parameters for efficient on-device learning. *Advances in Neural Information Processing Systems*, 33:11285–11297, 2020.

[12] François Chollet. *Deep Learning with R.*

[13] Bin Dai, Chen Zhu, and David Wipf. Compressing neural networks using the variational information bottleneck. 02 2018.

[14] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, et al. Tensorflow lite micro: Embedded machine learning for tinyml systems. *Proceedings of Machine Learning and Systems*, 3:800–811, 2021.

[15] Lei Deng, Guoqi Li, Song Han, Luping Shi, and Yuan Xie. Model compression and hardware acceleration for neural networks: A comprehensive survey. *Proceedings of the IEEE*, 108(4):485–532, 2020.

[16] Online On device MCU Transfer Learning. Vikram ramanathan.

[17] Ian J Goodfellow, Mehdi Mirza, Da Xiao, Aaron Courville, and Yoshua Bengio. An empirical investigation of catastrophic forgetting in gradient-based neural networks. *arXiv preprint arXiv:1312.6211*, 2013.

[18] Lennart Heim, Andreas Biri, Zhongnan Qu, and Lothar Thiele. Measuring what really matters: Optimizing neural networks for tinyml. *arXiv preprint arXiv:2104.10645*, 2021.

[19] Mark Horowitz. Computing's energy problem.

[20] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

[21] Norman Jouppi, Cliff Young, Nishant Patil, and David Patterson. Motivation for and evaluation of the first tensor processing unit. *IEEE Micro*, 38(3):10–19, 2018.

[22] Heechul Jung, Jeongwoo Ju, Minju Jung, and Junmo Kim. Less-forgetting learning in deep neural networks. *arXiv preprint arXiv:1607.00122*, 2016.

[23] Quoc V Le. Building high-level features using large scale unsupervised learning. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 8595–8598. IEEE, 2013.

[24] Davide Maltoni and Vincenzo Lomonaco. Continuous learning in single-incremental-task scenarios. *Neural Networks*, 116:56–73, 2019.

[25] Tom M. Mitchell. *Machine Learning: A multistrategy approach.*

[26] James O' Neill. An overview of neural network compression.

[27] Eugeniu Ostrovan. Tinyml on-device neural network training.

[28] Lorenzo Pellegrini, Gabriele Graffieti, Vincenzo Lomonaco, and Davide Maltoni. Latent replay for real-time continual learning. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 10203–10209. IEEE, 2020.

[29] Haoyu Ren, Darko Anicic, and Thomas A Runkler. Tinyol: Tinyml with online-learning on microcontrollers. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2021.

[30] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[31] Bharath Sudharsan, Piyush Yadav, John G Breslin, and Muhammad Intizar Ali. Train++: An incremental ml model training algorithm to create self-learning iot devices. In *2021 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/IOP/SCI)*, pages 97–106. IEEE, 2021.

[32] Alexey Tsymbal. The problem of concept drift: definitions and related work. *Computer Science Department, Trinity College Dublin*, 106(2):58, 2004.

[33] Joost van Amersfoort, Milad Alizadeh, Sebastian Farquhar, Nicholas Lane, and Yarin Gal. Single shot structured pruning before training. *arXiv preprint arXiv:2007.00389*, 2020.

[34] Gido M Van de Ven and Andreas S Tolias. Three scenarios for continual learning. *arXiv preprint arXiv:1904.07734*, 2019.

[35] Darrell Whitley, Timothy Starkweather, and Christopher Bogart. Genetic algorithms and neural networks: Optimizing connections and connectivity. *Parallel computing*, 14(3):347–361, 1990.

[36] Friedemann Zenke, Ben Poole, and Surya Ganguli. Continual learning through synaptic intelligence. In *International Conference on Machine Learning*, pages 3987–3995. PMLR, 2017.

[37] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. Hello edge: Keyword spotting on microcontrollers. *arXiv preprint arXiv:1711.07128*, 2017.

[38] J.M. Zurada. Analog implementation of neural networks. *IEEE Circuits and Devices Magazine*, 8(5):36–41, 1992.

Abstract in lingua italiana

Tiny Machine Learning (TinyML) è l'ambito di ricerca in cui si combinano le soluzioni di apprendimento automatico con i vincoli stringenti del hardware embedded/IoT.
Pochi centinaia di kilobyte di RAM sono disponibili su hardware tiny e la frequenza di clock dei processori è nell'oridine dei KHz. Con risorse tanto limitate è una sfida far funzionare le soluzioni di apprendimento automatico, in particolare le reti neurali. Nonostante questo sono stati raggiunti notevoli risultati, ad esempio nella rilevazione di parole chiave in segnali audio (keyword spotting).
Le soluzioni vengono progettate e allenate su cloud e successivamente convertite per operare su hardware tiny, tuttavia solo l'inferenza è supportata.
L'allenamento on-device, invece, si riferisce all'abilità di adattare un modello direttamente sul hardware edge/embedded. L'allenamento on-device ha numerosi vantaggi tra cui risparmio di energia, riduzione della latenza e miglioramento della privacy. Attualmente l'allenamento on-device per hardware non specializzato non è supportato da alcun framework per TinyML. Anche la ricerca accademica in questo ambito è poca e focalizzata su singoli problemi specifici.
Questa tesi sviluppa un insieme di strumenti per l'allenamento di reti neurali on-device su hardware tiny che può convertire modelli in formato embedded. Sono supportate reti neurali convoluzionali e reti neurali feed-forward. La soluzione è stata testata su dataset standard e nei contesti di transfer learning, incremental learning e concept drift.

Parole chiave: TinyML, reti neurali, on-device learning

Acknowledgements

I feel the warmest gratitude and appreciation for all the people who have supported me over the past year while working on this thesis:
Professor Manuel Roveri, for his deeply knowledgeable insights as well as for inspiring me and having faith in my abilities. I have always left your office energized and motivated.
Massimo Pavan, for his dutiful and dedicated supervision and availability.
Armando Caltabiano, Pierpaolo Lento, and Alessandro Bassi of Truesense, for their technical expertise and professional advice. As well as financial support which has rendered it easy to focus on work, unburdened by day-to-day practicalities.
Gabriele Viscardi, for his friendly hospitality and important technical knowledge.
My family and friends, who are always there. I hope to make you proud.